LIFE CYCLE INTEGRATION OF MODEL-BASED TESTING TOOLS

Mark Blackburn, Robert Busser, Aaron Nauman, Travis Morgan,
Systems and Software Consortium/T-VEC Technologies, Herndon, Virginia

Abstract

The paper discusses how organizations use specific model-based tools and evolved their existing engineering processes to develop and test large-scale critical applications. It discusses challenges and best practices observed from the use of model-based testing tools, and reflects on tool requirements that are essential for organizational adoption, including support for requirement-to-test traceability from requirement management tools, through requirement and design modeling, model-based test generation, to automated test execution and analysis using model-based testing tools that have qualification evidence to support use on safety-critical applications.

Introduction

A growing number of mission critical systems are being developed using model-based tools. These systems support complex modeling with simulation capabilities that help modelers better understand the dynamic aspects of the system, as well as code generation capabilities for various environments. In addition, users, customers, and organizations like the Federal Aviation Administration (FAA) and Food and Drug Administration (FDA) are concerned with many issues related to safety, certification, tool qualification, and how these tools fit into the overall life cycle of safety-critical systems development and verification.

Over the past several years, we have had the opportunity to work with different organizations, in various application domains, and been involved in the transition and adoption process of model-based testing into their organization. There are many requirements on the organization, users, and developers of the tools that appear almost mandatory as part of an effective adoption process. The completeness of the modeling environment is a critical element for adoption in many organizations. Although it often possible to test a subset of an application using model-based testing, organizations often resist adoption if a relatively complete solution is not available during pilot project trials. The modeling techniques and languages must be relevant to the applications under test (e.g., embedded systems with complex math, avionics, command and control, language processing, client-server), and support automated test execution against various languages in different environments. Finally, the learning curve must be relatively short, usually fewer than three month, but with rapid demonstration of the feasibility, usually within three days.

Scope

This paper is based on lessons learned from deploying various types of modeling capabilities since 1996. It uses a case study scenario, generalized from a company, to describe how organizations use specific tools to support requirement analysis, modeling, design for testability, and testing. It provides guidelines and recommendations observed from the use of model-based testing tools, and discusses the benefits, which include improved requirements and design, faster test failure analysis, better assessment of requirement-to-test completeness, and critical support for project measurement and management.

Life Cycle Support

The integrated environment generically referred to as the Test Automation Framework (TAF) integrates government and commercially available requirement and design modeling tools with test generation tools [1]. TAF integrates the DOORS® requirement management tool with the T-VEC Tabular Modeler (TTM) that supports the Software Cost Reduction (SCR) method [2] for requirement modeling. DOORS integrates also with Simulink®, which supports design-based models, and TAF integrates requirement models with design models to provide full traceability from the requirements source to the generated tests, as reflected in Figure 1. It integrates also with code

coverage tools produced by LDRA, Rational/IBM as well as open source tools such as GNU.

TAF has been used on many applications, some of which are reflected in Table 1 that cover software unit, integration and system-level testing. The models typically describe the functional requirements of a system or component, but describe also security requirements for a database. The target implementations range from web-based to embedded systems on various platforms and operating systems (OS), and test drivers (aka test scripts) that were generated to support automated test execution in many languages and data formats.

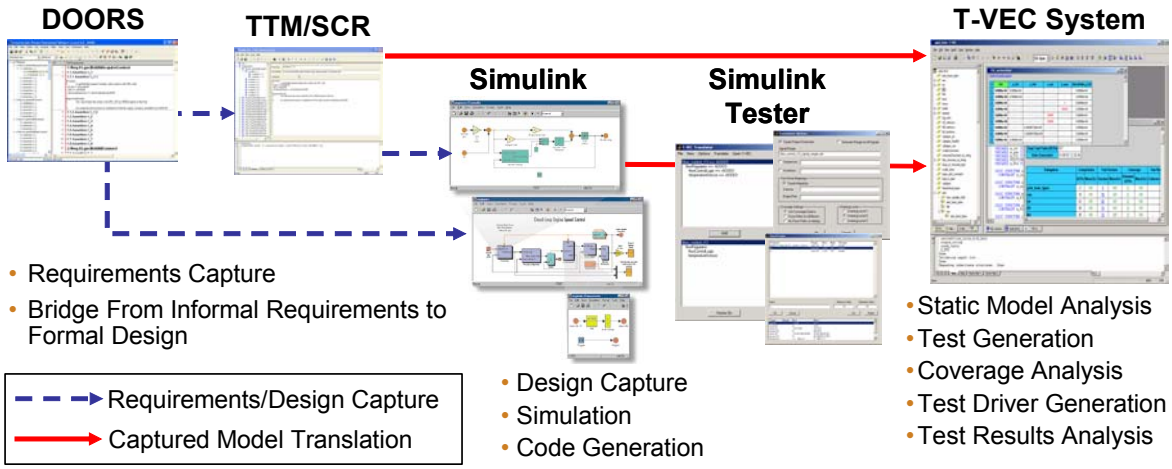

Figure 1. TAF Integrated Environment

Table 1. Application Summary

Application	Level	System/Component	OS	Test Langauge
Database security	System	Oracle	Win2K, XP	Perl/Java/JDBC
Smart card interoperability	System	Reference implementation	Win2K, XP	Java
SQL extension language processing	System	Parallel Database	Win2K	SQL extension
Copier/printer feature processing	System	Custom hardware	Unix	XML
Client-server web application	System	Web-to-database	Win2K/IE	Winrunner
Client-server	System	Tracking and certification	Win2k/CISC	DynaComm
Distributed billing system	System	Custom application	Unix	Perl
Umanned vehicles	System	Brake control	Unknown	VB-like
Mars polar lander	Software unit	Touch down monitor	Win2K	C
Command control for ship	System	System monitor	Unix	Slang script
Medical devices	Intergration	Mode switch	Unix/custom	C-like (custom)
Medical devices	Intergration	Monitoring and method selection	Custom	Custom
Medical devices	Intergration	Internal management	Unix/custom	C-like (custom)
Flight guidance mode logic	Unit/integration	Mode logic	Custom	Java
Avionics monitoring	System	Cruise energy management	Custom	SWAT (custom)
Mission management	System	Stores management	Custom	SWAT (custom)
Sonar	Unit/integration	Mode control	Custom	HTML
Utility	Unit	Transfer time conversion	Unix	C
Time card processing	System	Time card rules processing	Win2K, XP	Data file

Case Study and Problem Context

This case study discusses the application of TAF during a multi-year time period to create an engineering approach to model-based testing that starts with the system engineers that develop the requirement and interface specifications, to the design team that constructs more testable system and components, to the test engineering organization, quality assurance organization that interfaces with the certification authority, and the organization that develops and maintains that

engineering infrastructure. This case study provides details related to organizing models to support multi-team development and other related benefits.

The information presented in this section is generic. Several different companies' technical specifications were examined from Internet information and patent summaries to ensure the following information is presented in a product neutral form.

Like many companies that build high assurance life critical applications, zero defects is a requirement. The cost of the verification and validation (V&V) efforts for these companies often exceed fifty percent of the total effort, and the company discussed in this case study did confirm that its testing cost was significantly higher than fifty percent of the life cycle cost. In addition, the complexity of its systems continues to increase, along with greater scrutiny from the certification authorities such as the FAA and FDA, but the competitive market pressure means these high assurance requirements must be satisfied in increasingly shorter time periods.

This company uses advanced testing facilities including simulators, emulations, breadboard and hardware test environments, with comprehensive test analysis, measurement, tracking, reporting and logging capabilities. It is desirable to reduce the cost of testing, but schedule reduction is the most critical need in order to remain competitive in the market place. Most testing prior to the use of TAF was performed using manually produced test scripts that support fully automated test execution and results analysis. Even with this significant V&V support, the time and cost to create test cases (i.e., the test design process), and then implement those test designs into various scripting languages is labor intensive, time-consuming, and costly. The criticality of the systems requires them to perform comprehensive reviews of test procedures that can be several hundreds of lines of code. For any small product there can be over one thousand test scripts required to fully verify the product.

If a change is made to a product after it has been released to the field, for any regression testing need, the entire test suite must be re-executed. Often due to complex timing requirements, test scripts that might work for one release of a product might not work after a modification has been made to the system; such tests must be re-assessed, corrected or re-implemented, re-reviewed, and then re-executed. If common changes are required, such changes could require re-editing of hundreds of test scripts.

There are advantages to a comprehensive simulation and test infrastructure, but the sophisticated and the wide-spectrum set of application program interfaces (APIs) for controlling simulations sometimes provide far too many options for test designers and can lead to reduced robustness of the test scripts, especially since different simulation APIs have different timing characteristics; that is the timing of one sequence of API calls can vary by a few milliseconds from another set, even though they might achieve the same function.

Receiving certification approval is a critical and time-consuming part of product release. If the certification authority could be convinced that the TAF approach provides the verification rigor needed for certification and will allow the TAF verification results to be submitted as justification for approval, then the company will improve its ability to deliver complex systems with certification approval in a more cost effective manner.

Phased Implementation of TAF

This case study takes a chronological perspective as the integration of the entire model-based method that impacted many different organizations within this company. There were four distinct phases of involvement with this company. The effort started with a very small thread of functionality and transitioned into one of the most complex control mechanism that is common in many similar products. These successful demonstrations lead to the application of TAF on two different product-lines, and involved coordination with the design team, system engineering that wrote the product technical specification, test team, and the quality assurance organization involved in certification and tool qualification.

Many of the products this company develops tend to have a common data flow as conceptually represented in Figure 2. In real-time on a periodic basis, the product usually performs some sensing

function, while capturing information, and checks that information against some stored information within the product that is usually set by a user. Based on the information, usually collected and filtered over time, algorithms select options to issue responses or controls on the product. These products continue to evolve over time, and some users prefer different algorithms. It is common for functions such as **Check** to have many different types and combination of filtering, matching, and selection possibilities. A new combination is often called a feature when it is presented to a user, however the feature can impact many components within a system. This case study discusses the organizational and process impacts of developing a feature for the **Check** component that impacts **Filter**, **Match**, and **Select**.

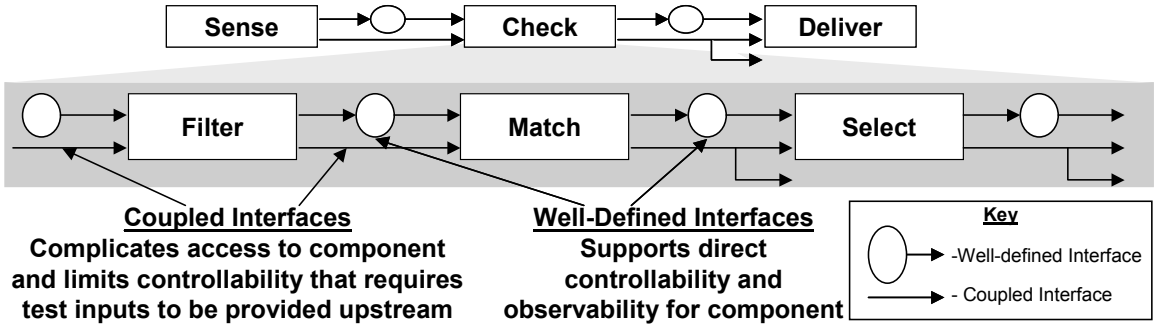

Figure 2. Conceptual Component of Example System

Prior to the engagement with the TAF team, as reflected in Figure 2, the components of the Check function were not partitioned with well-defined interfaces, rather the functionality was coupled, which made testing the functionality in each subcomponent (i.e., Filter, Match, and Select) more difficult. However, there is a verification requirement to demonstrate that every thread through a component or subcomponent is completely tested. Tight coupling makes this requirement difficult to achieve and demonstrate.

Phase I was a short pilot project effort to demonstrate the feasibility of applying model-based testing. During this phase, we applied the conceptual modeling process shown in Figure 3. We were able to quickly (i.e., approximately two hours) develop a model, map test drivers to the test environment, execute a test against a project, and find a minor problem with the memory mapping for the incoming message from the product. This encouraged the company to progress to Phase II.

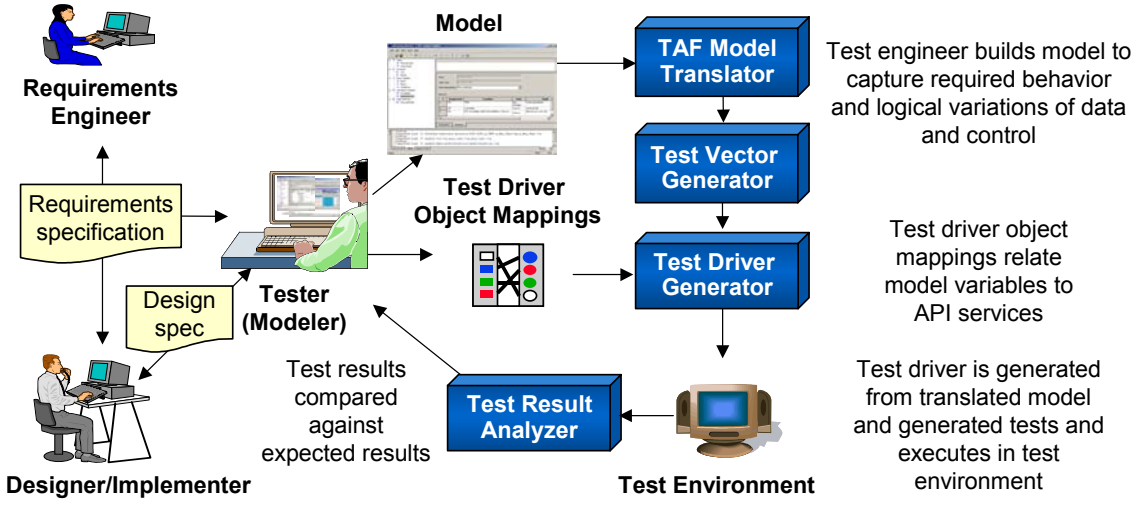

Figure 3. Modeling Process

Improved Test Infrastructure

Phase II was a challenging problem, because the model characterized a well-defined, but arguably one of the most complex control mechanisms of the entire product. We modeled approximately 130 requirement threads. The modeling process helped illustrate problems and anomalies, nothing serious, with documentation including the technical specification and interface specifications, which were maintained separately. However, the key issues arose when we created the object mappings to support automated test driver generation.

The test infrastructure was very robust, and used by both testers and developers. Test scripts were written in C++ in a Microsoft development environment and an extensive set of API services provided numerous ways to integrate different software versions simulators or target environments. The API services permitted program control for all interfaces to the product including the environment, but unfortunately, the API services had become feature rich with many overlapping functions. It was often difficult to understand how to initialize the simulation for a particular subsystem of features, and was difficult to uncover the functions necessary to setup consistent control of the simulator. These same problems plagued test engineers, especially less experienced engineers, because there were several hundred API services to support testing, and many different requirements for initializing different test environments, for various different types of product features.

We were able to work with key people that helped develop the test infrastructure and simulator to successfully build test drivers for the modeled requirements. We isolated the best services to accomplish the task and used those to produce the generated test drivers. More importantly, the efforts caused an initiative within the company to completely re-design the entire set of API services for the user community. The set of APIs were simplified and reduced down to about one quarter of the original number.

Requirement Analysis

The success of Phase II permitted us to move on to a new feature that was to be implemented in an existing product, which is referred to as Phase III. Originally this feature was going to be included in the next generation product, but a competitor was including this feature in their product. This market pressure forced this company to include this feature in an existing product.

The modeling started nearly in parallel to the design and implementation process. This permitted more continuous testing during development and allowed for early analysis of the technical specifications. We applied interface-driven requirement modeling that starts early during the requirement and design phase. This has been demonstrated to help in creating a more testable design and improving the requirement and interface specifications.

This company uses a two-phased release process of a technical requirement specification. During the first phase, the technical specification is under configuration control but can be evolved, reviewed and changed without official approval from a change control board. After the specification is released, a change control board must approve all changes, which increases the time and effort.

Fortunately, the modeling process started from a technical specification that had not been officially released. This was an exception to the typical testing process, because testing normally starts much later in the development process. However, during the process of modeling the requirement specification, about 100 specification problems were uncovered. All of these issues were discussed, and resolved, with the system engineers that developed the technical specification prior to the change control board. This intangible benefit of the model-based testing effort saved cost and effort by uncovering these issues, not to mention that the early resolution of the issues saved the designers time and effort from making potentially bad design choices due to issues in the requirement specification.

Design for Testability

Another issue that surfaced during Phase II was addressed during Phase III. As reflected in

Figure 2, the functionality in the existing system was tightly coupled due to numerous reasons related to power consumption and memory space limitations of the product. The interfaces between Filter, Match and Select were not well-defined. This complicated the testing process, requiring many tests to be initiated from higher levels in the system, such as Check because some of the inputs could be set upstream from the Check component. In addition, the outputs from the function such as Match were not visible. This made systematic and comprehensive testing of these lower-level components very difficult. Normally, to ensure coverage of the threads through the implementation of these lower-level components means increased testing from the high-level components, and sometime the number of tests can increase by an order of magnitude.

This effort started early enough that the designers were able to expose input and output interfaces, including internal state information to increase the testability significantly. Approximately 80 percent of the functionality was tested with improved interface support provided by the design team. The other 20 percent of the component elements could not be changed due to performance impacts, and re-testing effort. This significantly reduced the complexity of the model and tests, and provided greater test coverage with fewer tests to reduce time and cost.

This design for testability philosophy was applied to another product in Phase IV. The Phase IV effort involved another older product that was to be replaced by another new product, but again due to market pressure by a competing product, a feature was added to an older product. The success on Phase III provided substantial evidence for repeating the effort on Phase IV.

Modular Requirement Specifications

This company has some of the best technical specifications and interface documentation of any member company, however we identified during the modeling process a reason for organizing the specifications in a different way. As shown in Figure 4, the company documents the interface specification separate from the requirement specification. Although this is over simplified, conceptually one team member specified a model for the requirements in Section 1.1. A second team member made the model for Section 1.2, and another made a model for Section 1.3 of the requirement specification. The issue that emerged during the modeling process of Section 1.4, which describes feature interaction requirements between Filter, Match, and Select, is that many of these features described conditions that were already defined in a model. Because these modeled requirements had undergone the review process, and all the tests generated from these models were complete, with passing status, the decision was made to develop a separate model for the requirements of Section 1.4. The problem however, is that many overlapping or related conditions were already defined in the other models, and rather than reusing existing model elements, the choice was made to produce a separate model. In addition, models make it visible when related information that should be grouped together is separated.

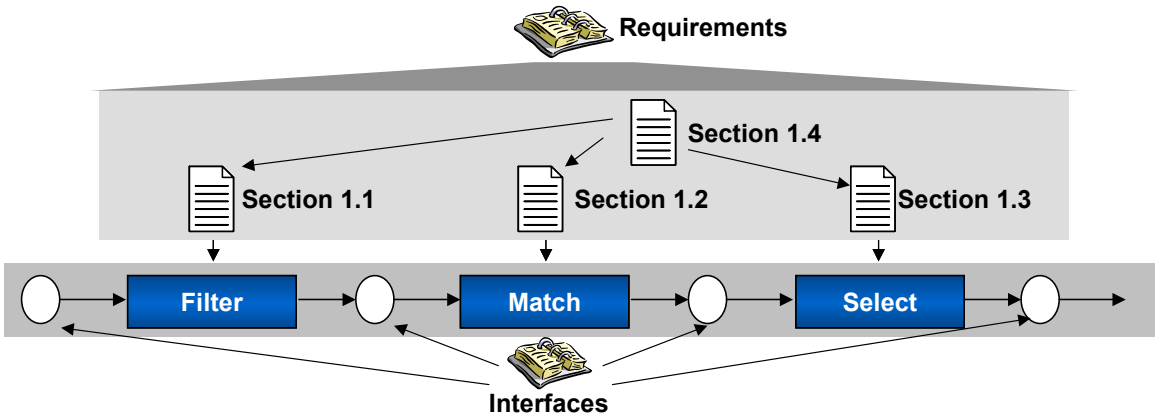

Figure 4. Organization of Requirement Feature

Fortunately, the TAF team had created, at the request of other TAF users, a model-include mechanism that allowed the Feature Interactions model, associated with Section 1.4, to include the Filter, Match, and Select models, so that existing model functionality could be reused, as shown in Figure 5. If conditions change in the future, the changes can and should be made in a single place. Just as it is a good practice to separate the interface specification in code (e.g., in a .h include file for the C programming language), it is a good practice to specify component interfaces separately. Figure 5 illustrates how common models represented interfaces separate from the required behavior. If the interface is related to the requirements, the interface model can be included with the model behavior. This practice is important, because if the interface changes, the changed interface is isolated in one place. Interface models tend to correspond with object mappings that represent the interface to the implementation. More details on common object mappings and test infrastructure are discussed below.

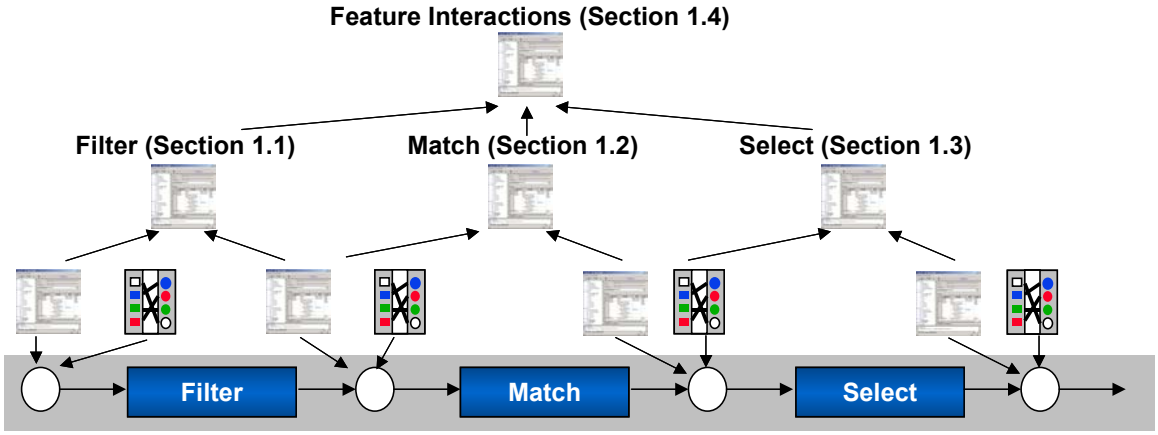

Figure 5. Models Represent Interfaces and Required Behavior

Finally, an important guideline, pointed out to the system engineers and specification team, is that the specification of the requirements should be associated with outputs used to assess the verification of the requirements. That means the requirement describing the interactions between components should be specified in the appropriate sections in Sections 1.1 through 1.3 as they relate to the interfaces of Filter, Match, and Select.

Model-Based Review Process

Companies that develop safety critical applications are often required to have code reviews as well as test procedure reviews. This company's existing process often required reviews of hundreds of test scripts that may have hundreds of lines of code. The new process relied on using validated TAF tools that satisfied the quality assurance organization's criteria for proper tool operation; this meant that the quality assurance organization believed that the tools would produce test vectors and test drivers that were complete and correct with respect to the model of the requirements. This permitted the review process to change.

The new process involved a review of the model by the system engineers to ensure the completeness and correctness with respect to the requirements. All requirements used the requirement traceability mechanism of the TAF tools to link the requirements to the generated test vectors. The second part of the process involved the designer/implementers, who reviewed the models and the associated test vectors presented in matrix format. The review of the model for the associated requirements that were directly traced to the requirements was much easier to understand and verify than the test drivers for the model.

Design decisions, implemented in code, result often in undocumented implementation-derived requirements. These implementation-derived requirements must be tested too. An important addition of the TAF process is that the designers were able to request the addition of a special type of model information called a "test constraint" for the

implementation-derived requirements. A test constraint results in additional tests, in addition to the requirement-based tests to support implementation-derived requirements. This further reduces the unit testing effort typically performed by the developer. Because tests were now being run in parallel to development, the implementers work effort was reduced. This however, would not have been possible without the designer providing additional test interfaces at the lower-level components.

The early interaction between the designers and test engineers improved the interfaces for testability, provided continuous testing earlier to reduce unit testing by the designers/implementers, and reduced the complexity of the testing to achieve more comprehensive test coverage with a reduced set of model-based tests.

Multi-team Model and Test Infrastructure

There are often significant skill and knowledge differences within an organization, and as discussed in previous sections, we relied on knowledgeable individuals to recommend simulation API services for scripting tests and for initializing the test environment in order to automate test execution. These knowledgeable individuals were better able to recommend specific services to carry out the functions to control the simulation. The test infrastructure for model based testing can be engineered to provide significant reuse of model interfaces and their associated object mappings, by leveraging the most knowledgeable resources within a company.

Figure 6 illustrates the two roles involved in the backend aspects of model-based testing. This is the process where modeled variables more closely related to the requirements must be mapped to the physical mechanisms that are used to set inputs (i.e., inject test inputs) and get outputs. The two roles include the Modeler and the Test Automation Architecture (who often plays a modeling role too). In this company, there was one test automation architect for all of the modelers supporting the Phase III and Phase IV effort. This one individual tailored and evolved the test driver schema to operate with two completely different testing environments and languages. The schema provided common reporting and execution mechanisms all based on the same framework, which is shown in Figure 7. The test automation architecture controls also the common object mappings that correspond to the common component interfaces. A modeler that might not know as much about the details on the test environment can focus on building the models from the requirements and then be directed to reference common object mappings in order to produce test drivers for all of the modeled functionality that works with the concrete interfaces of the actual target systems or simulation.

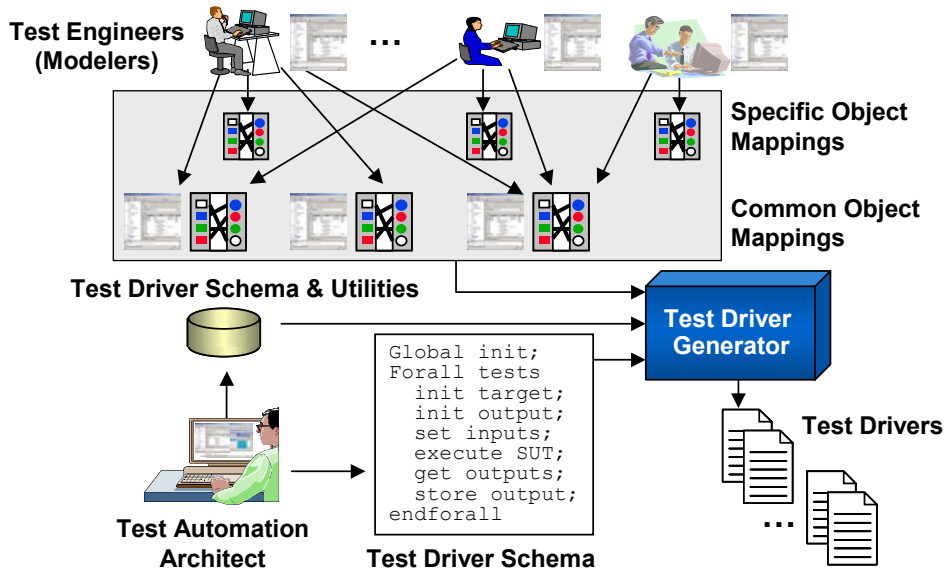

Figure 6. Roles in Test Driver Development

As shown in Figure 7 a modeler defines one object mapping for each output defined in a model. Within the object mapping, references are made to directory paths such as <HOME> that is assigned the project path (e.g., \TAF\course). From that <HOME> user-defined variable other information is referenced such as the location of the schema (e.g., <SCHEMA_HOME> = '<HOME>\test_driver_utilities'), which is where the test automation architecture provides common scripting utilities for reporting and logging, common initializations and declarations related to initializing the test environment, along with a common schema and common mapping file. The common mapping file (i.e., common.map) includes other common information that is pertinent to all modelers such as messages, literals, inputs, flags, and other variables. If an interface changes for some input, it is changed in one location (i.e., the inputs.map object mapping file), and all models that reference that input variable have a test driver interface that uses the API for setting that input variable. When such a change occurs, all test drivers can be regenerated to use the new interface. This avoids the problem in the current approach were every test script that references the input variable must be modified manually through some type of editing process.

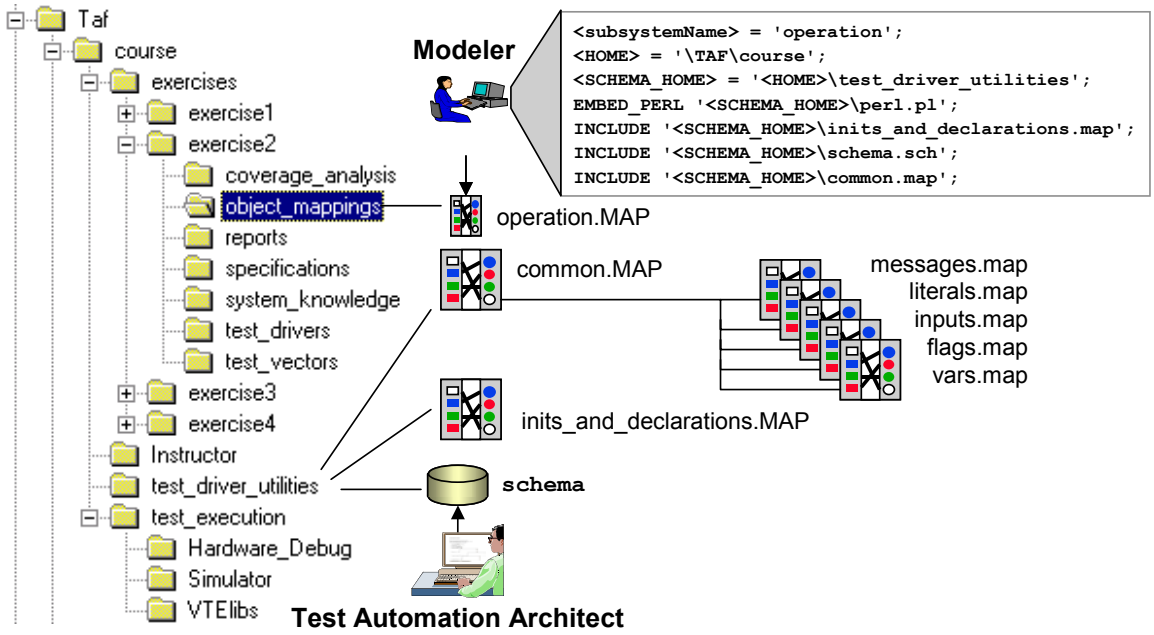

Figure 7. Test Infrastructure Organization

During both Phase III and Phase IV, the simulation environment, test infrastructure, component interfaces and reporting requirement continued to evolve. The test automation architect through updates to common object mappings, the schema, or utilities managed all of these changes in a way that was transparent to the other modelers.

Model-Based Measurement

During the beginning of Phase IV, which was about the middle of the Phase III project, we recognized that we had information and the need for using TAF measurement information to support project measurement [3]. Figure 8 provides a perspective on the key measurement information and how it relates to TAF requirements modeling. With this approach, there are four key base measures. System engineers are responsible for producing requirements, which results in the base measure number of requirements. A test engineer or modeler works in parallel with developers to refine requirements and build models to support iterative testing and development. Modeling introduces model variables, and this results in the base measure number of variables. After model translation and processing, the model requirements are converted into requirement threads, which is a base measure related to requirements. Finally, to support test driver generation, and test execution

and results analysis, the base measure number of object mappings is used.

This measurement-related information helped managers and project leads predict schedule duration and estimate project completion dates. Historical measurement information can be used prior to the start of a project, but it also is important to use data derived during the project.

Completed Project Ahead of Schedule

This company stated prior to the use of TAF that testing was more than 50% of their effort. Using model-based testing they completed the development and verification 9 weeks ahead of schedule. That had never occurred before.

Significant Reduction in Re-Testing

The company stated that after an experimental field trial of a product, they had to make changes and then completely re-test the entire product. They claimed that the TAF models and tests developed for the initial release, they were able to complete all testing 5 times faster as compared to their existing script-based testing process.

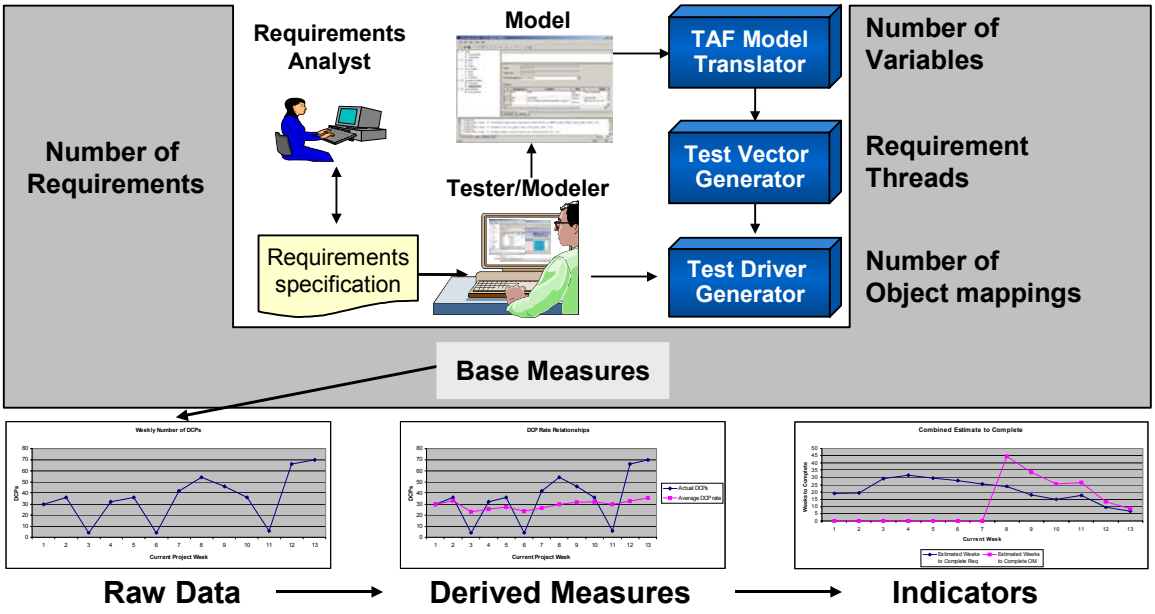

Figure 8. Process View of TAF Measurement

Guidelines and Recommendations

This section summarizes some of the key organizational and technical guidelines and recommendations provided in the case study.

1. Start with pilot projects to support organizational change. Stakeholders need to see quickly demonstrated evidence within their organization to commit to use model-based testing on a scheduled deliverable. It is often good to select a pilot project from a recently completed project, because the requirements are often well understood, even if not well documented. In addition, test cases and test results often exist that provide a more objective basis for comparison with the model-based tests.
2. Transition from a pilot project to a thread of an existing project. Select a thread that is likely to change often or have features extend it. The most leverage and benefits come from reusing and evolving one or more related models and the associated test infrastructure. Select a project prior to the requirement phase so that modeling can start early and help improve the requirements, while providing sufficient time to collaborate with the design team to improve the interfaces to support testability.
3. Start requirement modeling early to identify requirement defects sooner to reduce rework cost. Use interface driven modeling to ensure the component under test has testable interfaces. Define the requirements for each component in terms of the known interfaces.
4. Use goal-oriented modeling; work backwards by identifying each output at the component interfaces. Prioritize the ordering of the modeling for requirements thread to correspond with the expected development and/or integration of the component outputs associated with those requirement threads.

5. Identify and model interfaces separate from behavioral requirements to maximize the reuse and ensure a single point of definition for each modeled interface. Include model references to interfaces for components that are related to the functional requirements to ensures that the model of an interface is defined in one place. If changes occur to the interface, only one model will need to be changed.

6. Capture requirement traceability links in the requirement models. This provides important information to improve the review process. Tracing the requirements helps also in assessing the completeness of the model with respect to a requirement and related specification documents. Early modeling can identify incompletenesses in requirement documents that can be corrected early providing better input to designers and implementers.

7. Ensure requirement models capture negative cases as well as the positive cases of a requirement. The negative case can often uncover problems such as the problem that is the likely cause of the Mars Polar Lander [4], but also represent important safety or security cases. Establish modeling practices such as naming conventions, the use of terms that can be reused throughout the model, the use of constants, and traceability links.

8. Model continuously and in parallel with development. This can reduce testing effort for designers and implementers, and better ensure the design is testable. This results in an evolving automated test suite that should be executed for every build (daily, bidaily, weekly, etc.) of the system. This supports early identification of bugs that might be introduced by a developer change, and it makes it easier to understand and isolate the specific changes that introduced a defect into the system.

9. Extend requirement-based test models by adding tests constraints to a model to support implementation-derived requirements identified by the designer or implementer to reduce the unit testing effort traditionally performed by implementers.

10. Leverage the expertise of test automation experts, who often understand the most robust set of services for interfacing with the test environment as well as details related to initialization.

11. Develop common object mappings that correspond to modeled interfaces. Ensure that the test driver schema isolates test environment specifics such as initialization and declaration that can be controlled by the test automation expert. Develop and evolve one test driver schema per environment. Coordinate effort through a lead test automation expert that leverages common logging, reporting, configuration management and measurement support. Ensure the test driver schema maps requirement identifiers to test scripts for more efficient test failure analysis.

12. For more complex systems, analyze the interfaces, API, and requirement dependencies to ensure proper design of models that should be associated with test driver object mappings. This can maximize the reuse of common interface models and object mapping definitions to reduce cost and maintenance, and can reduce effort related to test sequencing.

Conclusion, Results and Benefits

The case study provides examples that summarize some of the benefits of model-based testing. The following provides a few other member company remarks that provide some perspective on the tangible as well as the intangible ROI associated with model-based test engineering.

One company stated there are many tangible benefits from model-based testing, but surprisingly, there are several intangible ROI benefits. At the end of the pilot demonstration the process and the supporting test infrastructure was 80-90% complete and relatively stable to support all follow-on testing. In addition, they identified several requirements for the testing infrastructure that could further automate the process or change the underlying process for the organization. For example: once an automated test suite exists, it can be run each time a build of the system occurs, this allows bugs to be identified by the developers much earlier in the development process, and it makes it easier to understand the specific changes that introduced a defect into the system rather than waiting weeks or months before manual testing is performed.

Lockheed Martin has used the TAF for many years [5], and has contributed significantly to the evolution and usage requirements including this release citation:

1. Some of the areas we have applied the TAF technology are in the Vehicle Systems Flight Control Laws, the Mission Systems Middleware, Digital Radio Controls, Auto Logistics AFB Basing and Flight Ops, Branch Health and Mode Determination software testing, and Reliability Enhancement and Re-engineering Program (RERP) Design and Test. These applications have been applied to various elements of multiple programs that include JSF/F-35, F2, and C5. Prior pilot applications performed on T-50 and F-16 showed applicability to legacy programs and would be beneficial in future upgrades to existing programs.

2. Applications on future upgrades to existing program that have extensive tabular formatted requirements have been identified as highly adaptable to the TAF technology through the use of TTM and T-VEC.

3. On one application related to Flight Control LAWS (Safety Critical software) it was determined that the application of TAF would significantly reduce the test efforts related to each release of the software. Typically there would be 6-12 releases for each version of this software with a total savings greater than six million dollars just in the test effort portion. Even more important would be the reduction in schedule time for the releases, which would result in greater dollar savings related to other personnel supporting the efforts.

4. In one experience we discovered some critical errors (such as potential divide by zero) during the design effort that would not have been caught until the test phase of the development and in some cases may not have been detected until much later when these unique conditions were met. In another experience we discovered an error, early in the development cycle, with the code generation tool being used. Benefit analysis on software development shows that the cost of resolving these errors grows exponentially as they move through the development phases. The actual cost savings of these can only be imagined but definitely go into the millions for our type software.

Conclusion

Model-based testing impacts the entire life cycle, and has been demonstrated to apply to many types of applications, such as embedded systems, language processing, client-server and web systems, distributed processing systems, command, control and monitoring, information processing business logic (IT systems), database, security, smart cards, and life critical systems such as medical devices and avionics systems. It has helped many organizations improve the requirements, drive improvements in the design of the target system and simulators, improve the test infrastructure with design for testability, guide the creation of modular requirement specifications, and demonstrate the cost benefits of modeling requirements early, with significant reduction in cost and schedule.

We have helped organizations in the adoption process, including the structuring of a multi-team modeling and test infrastructure, recommended model-based review practices, created tailored training, and shown how to use model-based measurement for project management.

References

[1] Blackburn, M.R., R.D. Busser, A.M., Nauman, Interface-Driven, Model-Based Test Automation, CrossTalk, The Journal of Defense Software Engineering, May 2003.

[2] Alspaugh, T.A., S.R. Faulk, K.H. Britton, R.A. Parker, D.L. Parnas, and J.E. Shore. *Software requirements for the A-7E aircraft*, Tech. Rep. NRL/FR/5546-92-9194. Washington, D.C.: Naval Research Lab, 1992.

[3] Blackburn, M.R., R.D. Busser, A.M., Nauman, Objective Measures from Model-Based Testing, STAREAST, May 2004.

[4] Blackburn, M.R., R.D. Busser, A.M. Nauman, R. Knickerbocker, R. Kasuda, Mars Polar Lander Fault Identification Using Model-based Testing, Eighth IEEE International Conference on Engineering of Complex Computer Systems, December 2002.

[5] Kelly, V. E.L. Safford, M. Siok, M. Blackburn, Requirements Testability and Test Automation, Lockheed Martin Joint Symposium, June 2001.